An application-centred framework for distributed engineering applications

A.H. Olivier and G.C. van Rooyen
Department of Civil Engineering, University of Stellenbosch, 7600 Stellenbosch, South Africa
(bertie@sun.ac.za)

Summary

The conceptual structure of an application that can support the structural analysis task in a distributed collaboratory is described in (van Rooyen and Olivier 2004). The application described there has a standalone component for executing the finite element method on a local workstation in the absence of network access. This application is comparable to current, local workstation based finite element packages. However, it differs fundamentally from standard packages since the application itself, and its objects, are adapted to support distributed execution of the analysis task. Basic aspects of an object-oriented framework for the development of applications which can be used in similar distributed collaboratories are described in this paper. An important feature of this framework is its application-centred design. This means that an application can contain any number of engineering models, where the models are formed by the collection of objects according to semantic views within the application. This is achieved through very flexible classes Application and Model, which are described in detail.

The advantages of the application-centred design approach is demonstrated with reference to the design of steel structures, where the finite element analysis model, member design model and connection design model interact to provide the required functionality.

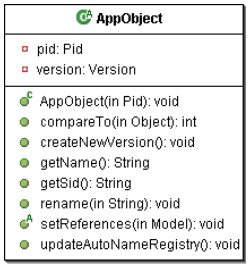
1 Introduction

With the development of new computer technologies and the availability of communication networks it has become possible to execute certain engineering tasks in distributed collaboratories. Traditional computer programs do not support the exchange of information at object level. This limits collaboration to the exchange of files via email or other protocols. The exchange of information at the level of objects dictates that both the application and the objects themselves have certain properties and capabilities (van Rooyen 2002). The requirements are:

- Objects are versioned.
- Objects report fundamental changes to themselves when they are used in consistent mode.
- Objects are capable of updating their state in response to a message from a collaborative source.
- Objects can represent themselves in a minimum size streamed format.
- The application is structured to establish runtime references on the basis of persistently identified objects.

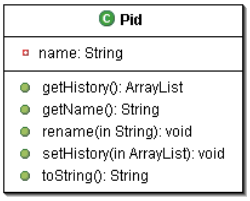
Key classes which are used to satisfy these requirements are described in the following section, followed by descriptions of the model and application classes. The convention is used throughout that engineering models are comprised of component objects. Reference will be made to collaboratory services, namely the database service, naming service, versioning service and consistency service. These are described in (van Rooyen and Olivier 2004) and (van Rooyen 2002).

2 Application objects



Class AppObject is the superclass of all application objects, and provide the basic functionality required of objects that are shared in the collaborative environment. An AppObject has an unique name in project space, a version, flags to indicate whether the object is used in consistent mode and whether is has undergone a fundamental change. It also has the ability to create a new version and forward a fundamental change notification to the project's consistency service. The subclasses have to be able to recognize fundamental changes and create a change notification when they are used in consistent mode. These aspects are described in the subsections below.

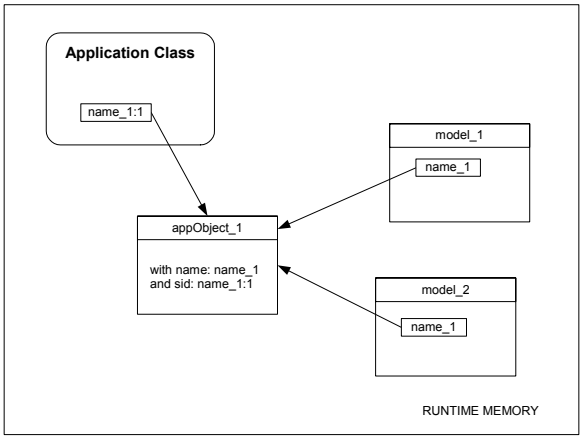
2.1 Persistently identified objects



Each AppObject has an unique name. These names are obtained from the collaboratory's naming service and are assigned to AppObjects when they are instantiated The name can be changed if necessary. The *name* of an AppObject is encapsulated in the persistent identifier (instance of class Pid) of the object. Using *names*, one can differentiate between AppObjects. However, it is not possible to distinguish between different versions of the same AppObject by referring to the *name*. Another identifier is introduced for the purpose of selecting a specific version of an AppObject. This is called the *selection identifier (sid)* . The *sid* is a concatenation of the *name* and the version *number*.

Names are used to map AppObjects within the specific context of a model. From this map it is possible to obtain the runtime reference of the particular version of the object used in that model if its *name* is known. The *modelMap* is maintained by the model.

Sids are used to map AppObjects in the runtime memory, independent of a specific model context. This provides the possibility to obtain a reference to an AppObject if its *name* and version *number* are known. The map is called the *applicationMap*, and it is maintained by the application.



Referencing of an AppObject in the runtime memory

2.1.1 Establishing references at runtime

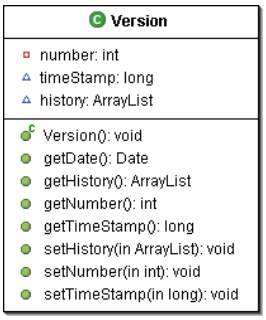
A key element of working in a distributed environment is the ability to restore relationships between objects once it is lost due to exchange of objects on an individual basis. If an AppObject knows the *name* or *sid* of another AppObject, it can obtain the runtime reference to that object from a registry that maps the *name* or *sid* to the object reference. The *name* is used to establish references within the context of a model, for example when an element needs the references of its nodes, as follows: (The element has the references of the nodes' pids).

```
public void setReferences(Model model){
  for (int i = 0; i < 3; i++)
  { node[i] = (Node)model.getComponent(nodePid[i].getName()) ;}}
```

The *sid* is used when a reference to an object in model_a is needed in model_b, for example when finite element, which is a component of a finite element model, is needed in a design model to provide section forces for design purposes.

2.2 Versioning of objects

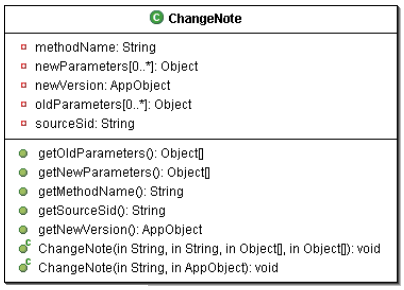
Each AppObject has a version and the functionality to recognise fundamental changes to itself. Versioning is controlled by the user to avoid a proliferation of versions. Creating a new version of an AppObject entails the following:



- The current version is updated using the collaboratory's versioning service.
- Since the object has a new version number, its sid changes and it has to be re-mapped in the applicationMap.
- Newly defined objects that were added to a model which is used in consistent mode are forwarded to other consistent users of the model at the point in time when these objects are versioned for the first time, and thereafter they are treated like new versions of normal objects used in consistent mode.
- If an object is used in consistent mode, the user who created the new version is released as a user of the previous version, except the special case of newly defined objects described directly above. The new version is registered as being used in consistent mode by the user. Its creator is initially the only consistent user of the newly versioned object, but the new version will also be offered to other consistent users of the previous version, and if they accept the new version, they become consistent users as well. Models, however, are treated differently: If a model has been checked out in consistent mode, that same model (sid) remains in consistent mode, even if the model version gets updated due to changes in the model. Otherwise complete models would have to be transferred due to version updating.

2.3 Updating an object

When an object which has been checked out of the project database for use in consistent mode undergoes a fundamental change, a ChangeNote is forwarded by the consistency service to all locations where it is used in consistent mode. An example of generating a ChangeNote is shown in the code snippet.

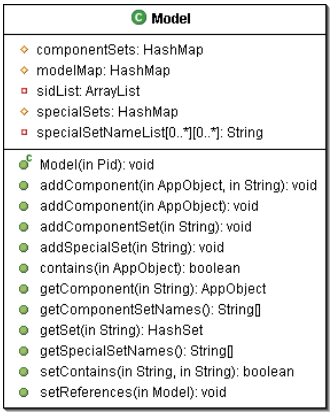



```
public void setCoordinates(double x1, double x2){
  if (x[0] != x1 || x[1] != x2){
    if(usedInConsistentMode && !updating){
        Double[] xold = {new Double(x[0]), new Double(x[1])};
        Double[] xnew = {new Double(x1),   new Double(x2)};
        reportFundamentalChange("setCoordinates", xold, xnew);}
    fundamentallyChanged = true ;
    x[0] = x1 ; x[1] = x2 ;}}
```

When a ChangeNote arrives at a workplace, the team member can analyse its content and decide whether to update his copy of the object or not. For this purpose the objects have an update method, for example as shown.

```
public void update(String methodName, Object[] newValues){
if (methodName.equals("setCoordinates")) {
   params = new Class[2];
   params[0] = double.class; params[1] = double.class;
   method = this.getClass().getMethod("setCoordinates", params);
   updating = true;
   method.invoke(this, newValues);
   updating = false;
```

3 Models



The basic structure of a model is that of a set of sets of component objects (Olivier 2002), where components are the essential elements of which the model consist. A finite element model, for example, will comprise of Node-components, Element-components, etc. However, a model can also contain special sets that are used, for example, to present certain views of the model, as well as special sets managed by the user. It also contains the tools that can create and edit components of the model, execute the algorithms of the given application, and present the results. A model is also an AppObject, so it is persistently identified and versioned. The component sets of a model are typically sets of instances of certain classes or interfaces, so that a component set references objects that are equivalent in some way. For example, a component set with the name "interface.INode" will reference all interface.INode instances in the model. This provides a mechanism to enumerate equivalent objects without searching for them. Component sets are added dynamically using the method `addComponentSet(String qualifierName)`. The qualifierName parameter is the name of a class or interface. When a component object is added to the model using the `addComponent(AppObject ao)` method, the object will be registered automatically in all the component sets for which it qualifies. When special sets are used, objects have to be added explicitly to them. The model maintains a map of all objects added to it in the modelMap (cf. section 2.1). The object names are used as keys, and references are established within the context of a model (cf. section 2.1.1).

The following code constructs a model with two component sets and adds a node and an element:

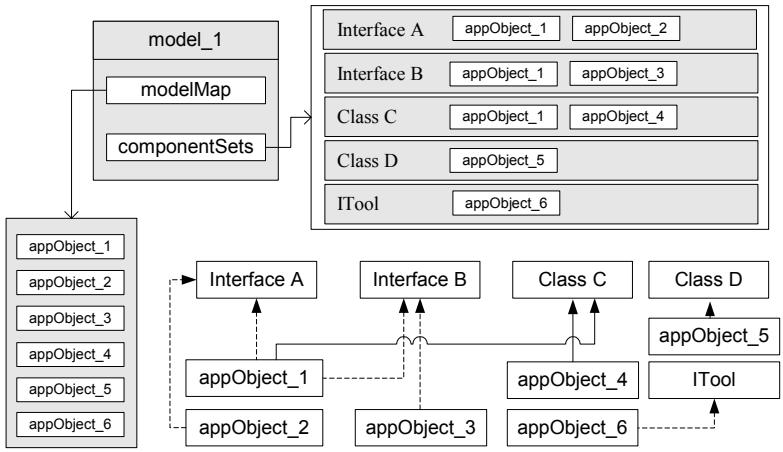
```
Model model = new Model(Pid.getPid("myModel"));
model.addComponentSet("interFace.INode");
model.addComponentSet("interFace.IElement");
model.addComponent(new Node(Pid.getPid("node1"));
model.addComponent(new FrameElement(Pid.getPid("element1"));
```

Note that since class Node implements the interface INode, the node object is registered in the interFace.INode component set, while class FrameElement implements interface IElement and the FrameElement instance is registered in the interFace.IElement set.

The figure shows a model that references six AppObjects. This model contains the following four component sets: “Interface A”, ”Interface B”, ”Class C” and “Class D”, and a toolset “ITool”.

AppObject_1 implements Interface_A, Interface_B and extends Class_C. Thus, it qualifies for the following component sets: “Interface A”, “Interface B” and “Class C”.

AppObject_6 is a tool which implements the ITool interface. Generally only one instance of a tool is required in the application. However, is is still referenced in the *modelMap*.



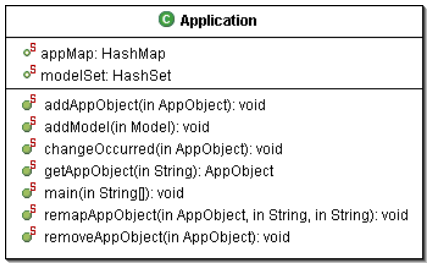
Data structure of a model instance

3.1 Distribution of models

A model can enumerate the selection identifiers of all the AppObjects that it contains. When a model is checked into / out of the project database, the *sids* of the objects registered in the model, as well as the names of the component sets that it contains are transferred. The objects referenced by the model are transferred only if the user chooses to do so. When a model is loaded at a workstation, the model is able to obtain the runtime references of it’s components from the *applicationMap* provided that the correct versions of the AppObjects are available in the memory. If not, the missing ones can be obtained from the project database. The user controls the actualization of references, which would typically be once all the AppObjects are available in the runtime memory.

4 Class Application

The Application class forms the cornerstone of the application (Olivier 2002). Its methods are all static, consequently they are available at any point by reference to class Application. All instances of AppObjects are mapped in the *applicationMap* using their *sids* as keys. Since the *sids* are used as keys, multiple versions of an AppObject can exist at the application level. This is necessary since the use of multiple models are supported, and in many practical cases different versions of objects are used in different models.



The application class also contains a static set, the *modelSet*, that references all the models in the application. Models can be added and removed at runtime, and the active model of the application can be set or obtained by an appropriate method call. This means that a user can easily switch his attention from one model to another.

Notifications of changes to objects used in consistent mode are distributed from the application to each model which references the affected object.

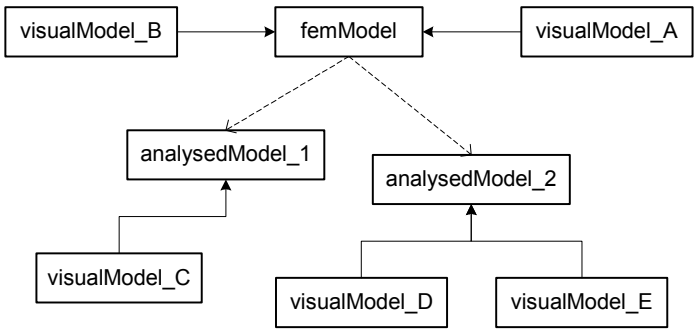
5 Using multiple models

The use of *sids* as keys in the *applicationMap* allows the use of multiple models in the application space without duplication of the underlying objects. This ensures consistency, at least within the local application. However, a specific version of an AppObject may be used by several models at once. When the user swithes from one model to another, the runtime references are set using the *modelMap* of the new active model, and some of the references in other models may be lost. Furthermore, certain transient information in objects have different values in different models. For example, the displacements of the same node will typically differ between different models in which the node is used. As a result a model cannot be displayed graphically if it is inactive.

This problem is overcome by allowing a user to take a snapshot of the active model before a new active model is selected. This snapshot is a new model which is a frozen (meaning not editable / not changeable) instance of a core model. The snapshot model duplicates both the fundamental and algorithmically dependent information of the core model from which it is extracted. In the example of finite element analysis the snapshot model is called an *AnalysedModel*. An AnalysedModel can be compared to a complete set of printed resultsheets since it can be used to create any possible graphical perspective, e.g. stresses, displacements, etc, but no changes can be made to the model. Snapshot models are useful to share results between users and also to maintain graphical views when switching between models on a single workstation.

In order to make it possible to have more than one view on a model, the *VisualModel* is introduced. A visual model comprises of AppObjects that serve as graphical representatives of the objects in the underlying model. An underlying model may have any number of visual models. The underlying model may be a core model, which is editable, in which case changes to the core model are made by user actions on any of the visual models that are shown. If the underlying model is a snapshot model, its visual models can simultaneously display different results, but no changes can be made to the snapshot model.

The figure below shows one finite element (fem) core model with two views on it, namely visualModel_A and visualModel_B. Two different analysed models were derived from the core model. Each one of these models are displayed graphically with their own visual models. The analysed models have no connection to the core model after they have been extracted. Thus, futher changes to the core model will not be reflected in these models.



Core model, analysed model, visual model

6 Conclusion

Distributed engineering applications need a special structure and functionalities which are not available in applications that were developed for local use on a single workstation. The requirements of the special case of structural engineering, as well as their consequences to the structure of a distributed analysis application were considered in detail in (van Rooyen and Olivier 2004). Details of a number of key classes and concepts which conform to and support the structure of a distributed analysis application were described in this paper. The structure of models, and an application that can deal with multiple models were described. Working with multiple models is facilitated by the introduction of snapshot models, as described. The concept of a visual model was introduced to allow multiple views on an underlying model. While structural analysis is used to develop and test the structures and concepts, it is postulated that they are applicable to a broad class of engineering applications.

7 References

Olivier, AH. (2002) ; *Object-oriented finite element framework,* MscEng thesis, University of Stellenbosch, Department of Civil Engineering, Stellenbosch, 2002.

van Rooyen, GC (2002); *Structural analysis in a distributed collaboratory*, Dissertation, University of Stellenbosch, Department of Civil Engineering, Stellenbosch, 2002.

van Rooyen, G.C. and Olivier, A.H., *Notes on structural analysis in a distributed collaboratory.* Paper submitted for the 10th Int. Conf. on Computing In Civil and Building Engineering, Weimar, Germany, 3-5 June 2004.